MST- Minimum Spanning Tree can be defined as a graph having 'n' nodes and 'n-1' edges. Also, all nodes should be reachable from each other. The sum of all the edges should be minimum. A graph can have multiple spanning trees that's why we have to choose the spanning tree with minimum sum of edges.

Kruskal's algorithm is extremely straightforward given that we know the concept of Disjoint Set Union (DSU).

We follow the below steps for find MST using Kruskal algo.

- Create a list of edges where each element will have three properties - node_val, parent_val, edge_weight → O(N+E)
- Initialize DSU data structure with "V" as number of nodes/vertices.
- Sort the edge list on the basis of weights of edges. (M log M) where M is size of list.

- Now, we just loop through the edge list and if both the nodes of current edge don't belong to same set we add weight of that edge to our total weight of MST and unionize the nodes. Otherwise, we just continue. O(M × 4 × alpha × 2)

```java
class Solution {
    static int spanningTree(int V, int E, List<List<int[]>> adj) {
        // Code Here.
        List<Edge> edges = new ArrayList<>();
        // O (N + E)
        for (int i = 0; i < V; i++) {
            List<int[]> ngs = adj.get(i);

            for (int[] ng: ngs) {
                int ngNode = ng[0], wg = ng[1];
                edges.add(new Edge(i, ngNode, wg));
            }
        }

        // O ( E log E)
        Collections.sort(edges, (a, b) -> Integer.compare(a.weight, b.weight));

        int res = 0;

        DSU dsu = new DSU(V);
        // O (E * 4 * alpha)
        for (Edge e: edges) {
            int src = e.src, dest = e.dest, wg = e.weight;
            if (dsu.sameComponent(src, dest))
                continue;
            dsu.merge(src, dest);
            res += wg;
        }

        return res;
    }

    private static class Edge {
        int src;
        int dest;
        int weight;

        Edge (int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }
}
```

```java
class DSU {
    int[] parent;
    int[] rank;

    DSU(int n) {
        this.parent = new int[n];
        for (int i = 0; i < n; i++)
            this.parent[i] = i;
        this.rank = new int[n];
    }

    int find(int x) {
        if (parent[x] == x)
            return x;
        return parent[x] = find(parent[x]);
    }

    void merge(int x, int y) {
        int ultX = find(x), ultY = find(y);
        if (rank[ultX] > rank[ultY]) {
            parent[ultY] = ultX;
        } else if (rank[ultY] > rank[ultX]) {
            parent[ultX] = ultY;
        } else {
            parent[ultY] = ultX;
            rank[ultX]++;
        }
    }

    boolean sameComponent(int x, int y) {
        return find(x) == find(y);
    }
}
```

Time Complexity: O(N+E) + O(E logE) + O(E*4α*2) where N = no. of nodes and E = no. of edges. O(N+E) for extracting edge information from the adjacency list. O(E logE) for sorting the array consists of the edge tuples. Finally, we are using the disjoint set operations inside a loop. The loop will continue to E times. Inside that loop, there are two disjoint set operations like findUPar() and UnionBySize() each taking 4 and so it will result in 4*2. That is why the last term O(E*4*2) is added.

Space Complexity: O(N) + O(N) + O(E) where E = no. of edges and N = no. of nodes. O(E) space is taken by the array that we are using to store the edge information. And in the disjoint set data structure, we are using two N-sized arrays i.e. a parent and a size array (as we are using unionBySize() function otherwise, a rank array of the same size if unionByRank() is used) which result in the first two terms O(N).